DS – Skúška otázky

1. **(15 bodov) Vysvetlite spôsob šírenia údajov v peer-to-peer systémoch pomocou klebetenia (gossiping). Ako nám tento spôsob môže pomôcť pri objavovaní konkrétnych služieb v neštruktúrovanom peer-to-peer systéme?**

Klebetenie (gossiping) je **decentralizovaný spôsob šírenia informácií**, ktorý funguje na princípe **epidemického správania**. Každý uzol v systéme si v pravidelných intervaloch **náhodne vyberie iného suseda** a vymení si s ním informácie, ktoré pozná. Týmto spôsobom sa informácie šíria **postupne, paralelne a veľmi rýchlo**, podobne ako vírus v populácii.

Výhodou gossiping protokolov je ich **vysoká škálovateľnosť a odolnosť voči zlyhaniam** – systém funguje aj v prípade, že niektoré uzly vypadnú alebo sú nedostupné. Nevýhodou je, že **neposkytujú striktné garancie doručenia**, ale v praxi dosahujú veľmi vysokú pravdepodobnosť rozšírenia informácie.

V **neštruktúrovaných peer-to-peer systémoch**, kde uzly nemajú globálny prehľad o sieti, sa gossiping používa na **objavovanie služieb alebo zdrojov**. Uzol nemusí poznať centrálny register – postupne získava informácie o existujúcich službách od svojich susedov. Po opakovaných výmenách sa informácia o službe rozšíri do celej siete a služba sa tak „objaví".

➡️ Tento prístup je vhodný najmä tam, kde sa sieť často mení a nie je možné udržiavať centrálnu evidenciu.

2. **(15 bodov) Popíšte problém dôvery (trust) v distribuovaných systémoch. Vysvetlite útok Sybil a možnosti ochrany voči nemu v decentralizovaných systémoch akými sú napr. blockchain.**

V distribuovaných systémoch je problém dôvery spôsobený tým, že **neexistuje centrálna autorita**, ktorá by overovala identitu a správanie jednotlivých uzlov. Uzly si musia navzájom dôverovať, aj keď môžu byť **neznáme, nespoľahlivé alebo útočné**.

Sybil útok nastáva vtedy, keď jeden útočník vytvorí **veľké množstvo falošných identít (uzlov)** a vystupuje ako viacero nezávislých účastníkov systému. Týmto spôsobom môže získať **neprimeraný vplyv na rozhodovanie**, napríklad pri hlasovaní, konsenze alebo prideľovaní zdrojov.

V **decentralizovaných systémoch**, ako je blockchain, sa proti Sybil útokom nebojuje priamym overovaním identity, ale tým, že sa **zvýši cena vytvorenia identity**. Používajú sa najmä tieto mechanizmy:

- **Proof-of-Work (PoW)** – vytvorenie identity alebo hlasovanie vyžaduje náročný výpočtový výkon,
- **Proof-of-Stake (PoS)** – vplyv uzla závisí od množstva vlastneného podielu (stake).

Tieto mechanizmy **nezabraňujú vytváraniu identít**, ale robia Sybil útok **ekonomicky nevýhodným**, pretože útočník by musel investovať veľké množstvo výpočtových zdrojov alebo kapitálu.

3. **(10 bodov) Škálovateľnosť môže byť dosiahnutá aplikáciou rôznych techník. Uveďte tieto techniky a vysvetlite ich.**

Škálovateľnosť distribuovaného systému znamená schopnosť systému **efektívne fungovať aj pri raste počtu používateľov, uzlov alebo požiadaviek**. Dosahuje sa najmä tromi základnými technikami:

1. **Ukrytie latencie (asynchrónna komunikácia)**
   Namiesto blokovania klienta počas komunikácie systém používa asynchrónne volania, vďaka čomu môže klient pokračovať v práci, kým čaká na odpoveď. Typickým príkladom je webový prehliadač, ktorý zobrazuje stránku postupne.
2. **Rozdelenie a distribúcia (partitioning)**
   Dáta alebo služby sa rozdelia na menšie časti a umiestnia na rôzne uzly. Každý uzol spracúva len časť záťaže. Príkladom je DNS, kde sú domény rozdelené do zón, pričom každú zónu spravuje iný server.
3. **Replikácia a cachovanie**
   Často používané dáta sa kopírujú bližšie k používateľom, čím sa znižuje latencia a záťaž centrálneho servera. Replikácia zvyšuje dostupnosť a cachovanie zlepšuje výkon.

Tieto techniky sa často kombinujú, aby sa dosiahla **vysoká dostupnosť, výkon a odolnosť systému**.

4. **(10 bodov) Načrtnite návrh viacvláknového servera, ktorý podporuje viacero aplikačných protokolov používajúc sockety ako rozhranie transportnej vrstvy nad operačným systémom.**

Vhodným riešením je návrh servera podľa **modelu Dispatcher–Worker**.

Server obsahuje **centrálne dispatcher vlákno**, ktoré:

- počúva na viacerých socketoch (portoch),
- prijíma prichádzajúce požiadavky,
- identifikuje použitý aplikačný protokol (napr. podľa portu alebo hlavičky správy).

Po prijatí požiadavky dispatcher **odovzdá spracovanie worker vláknu** z poolu pracovných vlákien. Každé worker vlákno spracuje konkrétnu požiadavku podľa príslušného protokolu a odošle odpoveď klientovi.

Tento prístup umožňuje:

- **paralelné spracovanie viacerých klientov,**
- podporu **rôznych aplikačných protokolov,**
- efektívne využitie systémových zdrojov.

Model je dobre škálovateľný a bežne sa používa v moderných serverových aplikáciách.

5. **(10 bodov) Vysvetlite prečo úplne usporiadaný multicast používajúci Lamportove logické hodiny nie je dobre škálovateľný?**

Úplne usporiadaný multicast (total order multicast) zabezpečuje, že **všetky procesy v skupine doručia správy v rovnakom poradí**. Pri použití **Lamportových logických hodín** sa každá správa označí časovou značkou a procesy si správy triedia podľa týchto značiek.

Problém so škálovateľnosťou spočíva v tom, že aby bol poriadok správ jednoznačný a konzistentný, **každý proces musí vedieť, že žiadna správa s menšou časovou značkou už nepríde**. To sa v praxi dosahuje tým, že:

- procesy si **navzájom potvrdzujú (ACK)** prijatie správ,
- správa môže byť doručená až vtedy, keď je isté, že ju poznajú všetky procesy.

Počet potvrdzujúcich správ rastie s počtom procesov **kvadraticky – $O(N^2)$**. Pri veľkom počte uzlov to vedie k:

- zahlteniu siete,
- zvýšenej latencii,
- komunikačným úzkym miestam (bottleneckom).

Preto je total order multicast založený na Lamportových hodinách **nevhodný pre veľké a dynamické systémy**.

6. **(15 bodov) Vysvetlite koncept Lamportových logických hodín a jeho využitie v úplne usporiadanom skupinovom vysielaní. Vysvetlite ako umožňujú vektorové hodiny zachytiť kauzalitu medzi vysielanými správami.**

Odpoveď:

Lamportove logické hodiny sú mechanizmus na **určenie poradia udalostí** v distribuovanom systéme bez globálneho fyzického času. Každý proces si udržiava čítač, ktorý:

- zvyšuje pri každej lokálnej udalosti,
- zvyšuje pri odoslaní správy,
- pri prijatí správy sa nastaví na maximum lokálneho času a času zo správy + 1.

Lamportove hodiny zabezpečujú, že ak udalosť A **kauzálne predchádza** udalosti B, potom jej časová značka je menšia. Používajú sa napríklad pri **total order multicaste**, kde sa správy triedia podľa časových značiek (a ID procesu pri rovnosti).

Nevýhodou je, že Lamportove hodiny **nevedia rozlíšiť, či dve udalosti spolu súvisia kauzálne alebo sú nezávislé**.

Tento problém rieši **vektorové hodiny**. Každý proces si udržiava **vektor časov**, kde každá položka reprezentuje známy stav ostatných procesov. Vektorové hodiny umožňujú:

- presne určiť kauzálny vzťah medzi udalosťami,
- zistiť, či sú udalosti **kauzálne závislé alebo súbežné (concurrent)**.

Nevýhodou vektorových hodín je ich **pamäťová a komunikačná náročnosť**, keďže veľkosť vektora rastie lineárne s počtom procesov.

7. **(15 bodov) Vysvetlite čo je to konsenzus v skupine replikovaných procesov v distribuovanom systéme odolnom voči svojvoľným/byzantským poruchám (arbitrary/byzantine failures) a aký je dôvod pre jeho zavedenie. Vysvetlite konsenzuálny protokol PBFT?**

Konsenzus je problém, pri ktorom sa skupina replikovaných procesov musí **dohodnúť na jednej spoločnej hodnote alebo poradí operácií**, a to aj v prípade, že niektoré procesy zlyhajú alebo sa správajú chybne.

V prostredí s **byzantskými poruchami** môžu uzly:

- posielať nesprávne správy,
- správať sa nepredvídateľne,
- alebo úmyselne klamať.

Preto je konsenzus nevyhnutný na zachovanie **integrity dát a konzistencie systému**.

PBFT (Practical Byzantine Fault Tolerance) je konsenzuálny protokol, ktorý:

- toleruje až **f byzantských uzlov**, ak je v systéme aspoň **3f + 1 replík**,
- používa viacfázovú výmenu správ (pre-prepare, prepare, commit),
- zabezpečuje, že všetky korektné uzly sa dohodnú na rovnakom poradí operácií.

PBFT nevyžaduje náročné výpočty (na rozdiel od PoW) a je vhodný pre **permissioned systémy**, kde je počet uzlov obmedzený.

8. **(10 bodov) Presne opíšte čo znamená pojem škálovateľný distribuovaný systém.**

Distribuovaný systém je **škálovateľný**, ak dokáže **zvládať rast záťaže** (počtu používateľov, požiadaviek alebo uzlov) **bez výrazného zhoršenia výkonu**.

Rozlišujeme tri základné dimenzie škálovateľnosti:

1. **Škálovateľnosť veľkosti** – schopnosť systému rásť pridaním nových používateľov alebo uzlov.
2. **Geografická škálovateľnosť** – schopnosť fungovať efektívne aj pri fyzickom rozptýlení komponentov.
3. **Administratívna škálovateľnosť** – schopnosť systému fungovať naprieč viacerými nezávislými administratívnymi doménami.

Škálovateľný systém sa vyhýba centralizovaným službám, dátam a algoritmom, ktoré by sa stali úzkym miestom, a využíva techniky ako **asynchrónna komunikácia, replikácia, cachovanie a distribúcia záťaže**.

9. **(10 bodov) Nech klient volá asynchrónnym RPC server a následne čaká, kým server nevráti odpoveď pomocou iného asynchrónneho RPC volania. Vysvetlite, či je tento prístup rovnaký ako nechať klienta volať normálne RPC. Čo by sa stalo, ak nahradíme asynchrónne RPC volania jednosmernými (one-way) RPC volaniami?**

Pri **normálnom (synchronnom) RPC** je klient po odoslaní požiadavky **blokovaný**, kým server nevráti odpoveď. Klient nemôže pokračovať v práci a počas čakania nevyužíva efektívne čas.

Pri **asynchrónnom RPC** klient odošle požiadavku a **pokračuje v ďalšej činnosti**. Odpoveď zo servera je doručená neskôr, napríklad pomocou **callbacku alebo samostatného asynchrónneho volania**. Ak klient po odoslaní asynchrónnej požiadavky aktívne čaká na odpoveď, správanie sa môže javiť podobne ako pri normálnom RPC, ale rozdiel je v tom, že:

- komunikačný kanál nie je blokovaný,
- klient môže medzičasom vykonávať inú prácu.

Ak by sme namiesto asynchrónnych RPC použili **jednosmerné (one-way) RPC**, klient by **nedostal žiadnu odpoveď**. To znamená, že:

- klient nevie, či server požiadavku prijal,
- nevie, či bola úspešne spracovaná,
- stráca sa možnosť detekcie chýb.

One-way RPC je preto vhodné len pre operácie, kde **nezáleží na výsledku** alebo kde je výsledok riešený iným mechanizmom.

10. **(10 bodov) Majme neblokujúci primary-backup protokol pre zabezpečenie sekvenčnej konzistencie v distribuovanom systéme. Poskytuje takýto distribuovaný systém vždy read-your-writes konzistenciu? Svoju odpoveď zdôvodnite.**

Primary-backup protokol funguje tak, že:

- všetky zápisy sú vykonávané na **primárnom uzle**,
- zmeny sú následne replikované na **záložné (backup) uzly**.

Aj keď takýto systém zabezpečuje **sekvenčnú konzistenciu**, **nezaručuje vždy read-your-writes konzistenciu**.

Problém nastáva v situácii, keď:

- klient zapíše hodnotu na primárny uzol,
- následne číta hodnotu z backup uzla,
- ale replikačná aktualizácia sa ešte nestihla preniesť.

V takom prípade môže klient **prečítať starú hodnotu**, aj keď jeho zápis bol úspešný. To znamená, že vlastnosť *read-your-writes* nie je splnená.

Read-your-writes konzistencia by bola zaručená iba vtedy, ak by klient:

- vždy čítal z primárneho uzla,
- alebo systém zabezpečil synchronizáciu replík pred čítaním.

11. **(15 bodov) Vysvetlite koncept spoľahlivej skupinovej komunikácie medzi replikovanými procesmi a možnosti jej škálovania. Popíšte fungovanie atómovej skupinovej komunikácie.**

Spoľahlivá skupinová komunikácia zabezpečuje, že správa odoslaná do skupiny je:

- doručená **všetkým korektným členom skupiny**, alebo
- nedoručená **nikomu**.

Tým sa zabezpečí konzistentný stav medzi replikovanými procesmi.

Pre škálovanie skupinovej komunikácie sa používajú techniky ako:

- hierarchické skupiny,
- multicastové stromy,
- zoskupovanie uzlov do menších podskupín.

Atómová skupinová komunikácia (atomic multicast) rozširuje spoľahlivosť o **úplné usporiadanie správ**. To znamená, že:

- všetky procesy prijmú správy,
- **v rovnakom poradí**.

Atómová komunikácia je základným stavebným prvkom pre:

- replikované databázy,
- stavové replikované služby,
- implementáciu konzistentných replikačných protokolov.

12. **(15 bodov) Vysvetlite problém konzistencie zameranej na údaje v distribuovanom systéme a uveďte konkrétne konzistenčné modely založené na usporadúvaní operácií. Aký je význam použitia modelu konečnej konzistencie (eventual consistency model) v praktických aplikáciách na rozdiel od silnejších modelov?**

Konzistencia zameraná na údaje (data-centric consistency) rieši otázku, **ako a v akom poradí sú operácie nad dátami viditeľné** pre rôzne procesy v distribuovanom systéme.

Medzi konzistenčné modely založené na usporadúvaní operácií patria:

- **Sekvenčná konzistencia** – všetky procesy vidia operácie v rovnakom poradí.
- **Kauzálna konzistencia** – zachováva príčinné vzťahy medzi operáciami.
- **Silná (striktná) konzistencia** – každé čítanie vidí najnovší zápis.

Tieto silnejšie modely však obmedzujú výkon a škálovateľnosť systému.

Eventual consistency povoľuje dočasné nekonzistencie medzi replikami, ale garantuje, že:

- ak už nepribúdajú nové zápisy,
- všetky repliky sa časom zosynchronizujú.

V praxi sa používa preto, že:

- umožňuje vysokú dostupnosť,
- znižuje latenciu,
- je vhodná pre veľké distribuované systémy (napr. DNS, cloudové databázy).

13. **(10 bodov) Majme distribuovaný systém, ktorý podporuje replikáciu objektov, v ktorej sú všetky volania metód úplne usporiadané. Predpokladajme tiež, že vyvolanie objektu je atómické (napr. pretože každý objekt automaticky lockne keď sa volá jeho metóda). Poskytuje takýto systém vstupnú konzistenciu? A čo sekvenčná konzistencia?**

Ak sú **všetky volania metód úplne usporiadané (total order)**, znamená to, že **všetky procesy vidia volania v rovnakom poradí**. Ak je zároveň vyvolanie objektu **atómické** (napríklad vďaka automatickému uzamknutiu objektu pri volaní metódy), systém **poskytuje sekvenčnú konzistenciu**. Každý proces teda pozoruje rovnakú sekvenciu operácií nad objektmi.

Takýto systém však **automaticky neposkytuje vstupnú (entry) konzistenciu**. Entry konzistencia vyžaduje, aby bol prístup ku konkrétnej dátovej položke možný **až po explicitnej synchronizácii viazanej na túto položku** (napríklad získaním zámku). Samotné úplné usporiadanie volaní metód nestačí – systém by musel explicitne riadiť, ku ktorým dátam sa synchronizácia viaže.

➡️ Zhrnutie:

- **Sekvenčná konzistencia: ÁNO**
- **Entry konzistencia: NIE (bez dodatočných synchronizačných mechanizmov)**

14. **(10 bodov) Predpokladajme, že by ste mohli použiť iba neblokujúce asynchrónne komunikačné operácie na posielanie a prijímanie správ. Ako by ste implementovali operácie na blokujúce synchronné komunikačné operácie pomocou týchto asynchrónnych operácií?**

Blokujúcu synchronnú komunikáciu je možné implementovať nad neblokujúcimi asynchrónnymi operáciami tak, že po odoslaní správy **proces aktívne čaká na odpoveď**.

Postup je nasledovný:

1. Proces odošle správu pomocou neblokujúcej operácie `send`.
2. Následne opakovane kontroluje, či odpoveď už dorazila, napríklad:
   - pomocou slučky s neblokujúcim `receive`,
   - alebo pomocou callbacku, eventu či signalizačného mechanizmu.
3. Proces je zablokovaný až do momentu, keď odpoveď príde.

Z pohľadu aplikácie sa takáto komunikácia správa **ako synchronná**, aj keď je implementovaná pomocou asynchrónnych primitív. Efektívnejšie riešenia využívajú **udalosti alebo callbacky**, aby sa predišlo neefektívnemu aktívnemu čakaniu (polling).

15. (10 bodov) Súbor je replikovaný na 10 serveroch. Vymenujte všetky kombinácie kvóra na čítanie a kvóra na zápis, ktoré povoľuje hlasovací algoritmus replikačného protokolu založeného na hlasovacom kvóre.

Pri hlasovacom kvórovom replikačnom protokole musia byť splnené tieto podmienky:

- **NR + NW > N**, aby sa čítanie a zápis vždy prekrývali,
- **NW > N / 2**, aby sa zápisy navzájom prekrývali.

Pre **N = 10** platí:

- **NW > 5** → minimálne **NW = 6**.

Možné kombinácie (NR, NW) sú napríklad:

- (1, 10)
- (2, 9)
- (3, 8)
- (4, 7)
- (5, 6)

Všetky tieto kombinácie zabezpečujú, že:

- čítanie vždy získa aspoň jednu aktuálnu repliku,
- zápisy sa nikdy navzájom „neprebijú".

----------Neúplné otázky----------

16. Konzistenčné modely (najlepší model pre email) – (strong, eventual..).

Emailový systém je typický príklad aplikácie, kde **nie je potrebná silná globálna konzistencia**, ale je dôležitá **konzistencia z pohľadu používateľa**.

Najvhodnejším prístupom je kombinácia:

- **slabej (eventual) konzistencie na úrovni systému**,
- a **client-centric konzistencie**, konkrétne **monotonic reads**.

Model **monotonic reads** zabezpečuje, že ak používateľ raz videl určitý stav svojej emailovej schránky, pri ďalších prístupoch (napr. z iného zariadenia) už neuvidí starší stav. Tým sa zabráni čítaniu zastaraných údajov zo starších replík.

Eventual consistency umožňuje:

- vysokú dostupnosť,
- prácu aj pri výpadkoch spojenia,
- synchronizáciu dát po obnovení siete.

➡️ Preto je emailový systém najlepšie riešiť **eventual consistency** + **monotonic reads**, čo poskytuje dobrý kompromis medzi používateľským komfortom a škálovateľnosťou.

17. Vysvetlite, čo je konsenzus v distribuovaných systémoch a opíšte, čo je Raft.

Konsenzus v distribuovaných systémoch je mechanizmus, pomocou ktorého sa **viaceré uzly dohodnú na jednej spoločnej hodnote alebo poradí operácií**, a to aj v prípade, že niektoré uzly zlyhajú. Je nevyhnutný najmä v systémoch s replikovanými dátami, aby všetky repliky zostali konzistentné.

Raft je konsenzuálny protokol určený pre systémy **odolné voči výpadkom (crash failures)**. Je navrhnutý tak, aby bol jednoduchší na pochopenie a implementáciu než Paxos.

Raft rozdeľuje uzly do troch rolí:

- **Leader** – prijíma požiadavky klientov a riadi replikáciu logu,
- **Followers** – pasívne replikujú log od leadera,
- **Candidates** – uzly, ktoré sa uchádzajú o rolu leadera.

Protokol funguje v troch hlavných častiach:

1. **Voľba leadera**,
2. **Replikácia logu**,
3. **Zabezpečenie bezpečnosti (safety)**.

Raft garantuje, že všetky korektné uzly majú **rovnaký log v rovnakom poradí**, čím zabezpečuje konzistenciu systému.

18. Čo je middleware a aký má zmysel ho používať?

Middleware je softvérová vrstva umiestnená **medzi operačným systémom a aplikáciami**, ktorá poskytuje spoločné služby pre distribuované systémy. Jej cieľom je **skryť heterogenitu a zložitosť distribúcie**.

Typické služby middleware zahŕňajú:

- **komunikáciu** (napr. RPC),
- **správu transakcií**,
- **bezpečnosť** (autentifikácia, autorizácia),
- **zotavenie z chýb** a spoľahlivosť.

Používanie middleware má zmysel, pretože:

- zjednodušuje vývoj aplikácií,
- zvyšuje prenositeľnosť a interoperabilitu,
- umožňuje opätovné použitie riešení naprieč systémami.

Príkladmi middleware sú **REST API, message brokery (Kafka, RabbitMQ), CORBA alebo gRPC**.

19. Máme synchronný blokujúci model komunikácie – ako ho spraviť asynchrónnym (pričom synchronné rozhranie má tiež existovať)?

Synchronnú blokujúcu komunikáciu môžeme prerobiť na asynchrónnu tak, že **samotnú komunikáciu vykonáme na pozadí**, zatiaľ čo rozhranie pre aplikáciu zostane nezmenené.

Možné riešenia sú:

- použitie **samostatných vlákien** na odosielanie a prijímanie správ,
- použitie **asynchrónneho RPC s callbackmi**,
- využitie **event-driven mechanizmov**.

Z pohľadu aplikácie sa volanie môže javiť ako synchronné, ale v skutočnosti:

- hlavné vlákno nie je blokované,
- odpoveď je spracovaná po jej príchode.

Takýto prístup kombinuje **jednoduchosť synchronného rozhrania** s **výkonnostnými výhodami asynchrónnej komunikácie**.

20. Či je potrebné odpovedať na každú správu pri Lamportovom čase, aby bol total order multicast – a popísať prečo.

Áno, pri implementácii **úplne usporiadaného multicastu pomocou Lamportových časových značiek** je potrebné **potvrdzovať prijatie každej správy**.

Dôvodom je, že proces môže správu doručiť až vtedy, keď má istotu, že:

- neexistuje žiadna iná správa s menšou časovou značkou, ktorá ešte len dorazí,
- všetky procesy poznajú aktuálny stav logických hodín.

Potvrdzovanie (ACK) zabezpečuje, že **fronty správ všetkých procesov sú konzistentné** a správy sú doručené v rovnakom poradí. Bez potvrdení by mohlo dôjsť k porušeniu úplného usporiadania.

Nevýhodou tohto prístupu je vysoká komunikačná réžia, čo negatívne ovplyvňuje škálovateľnosť.

21. Total ordering multicast – či je potrebné potvrdzovanie cez acknowledgements – zdôvodni.

Pri **total order multicaste** je cieľom zabezpečiť, aby **všetky procesy v skupine doručili správy v rovnakom poradí**. To je možné dosiahnuť rôznymi spôsobmi, a preto odpoveď nie je striktne „len áno" alebo „len nie".

- **Pri niektorých implementáciách** (napr. založených na Lamportových logických hodinách) sú **acknowledgements potrebné**, aby proces vedel, že ostatné uzly už prijali správu a že neexistuje iná správa s menšou časovou značkou, ktorá by ešte mohla prísť. Bez potvrdení by nebolo možné bezpečne rozhodnúť o poradí doručenia správ.
- **Pri iných implementáciách** (napr. s centrálnym sequencerom alebo pri použití konsenzuálneho protokolu) **acknowledgements nemusia byť explicitne potrebné**, pretože poradie správ je určené jedným uzlom alebo dohodou medzi uzlami.

➡️ Záver:
Acknowledgements nie sú principiálne nevyhnutné pre každý total order multicast, ale sú **často používané v decentralizovaných riešeniach**, kde neexistuje centrálna autorita určujúca poradie správ.

22. „time algorithm to check latency between systems“ – teda otázka na časový algoritmus na kontrolu latencie medzi systémami

Na meranie latencie medzi dvomi systémami v distribuovanom prostredí sa používajú **časové synchronizačné algoritmy**, ktoré pracujú s odhadom oneskorenia správ.

Jedným z klasických riešení je **Cristianov algoritmus**:

- klient pošle požiadavku serveru na aktuálny čas,
- server odpovie so svojím časom,
- klient odhadne **round-trip delay** a upraví svoj čas s predpokladom symetrickej latencie.

V praxi sa najčastejšie používa **NTP (Network Time Protocol)**, ktorý:

- opakovane meria oneskorenie medzi klientom a viacerými časovými servermi,
- filtruje extrémne hodnoty,
- odhaduje **latenciu aj časový offset**.

Tieto algoritmy umožňujú:

- približne zosynchronizovať hodiny,
- nepriamo získať informáciu o latencii medzi systémami.

➡️ Presné meranie latencie v distribuovaných systémoch nie je možné bez globálneho času, preto sa vždy pracuje len s **odhadmi**.